

第一章

线性结构：链表、栈与队列

当你走进一间杂乱无间的房间，找一本书可能要翻遍每个角落，这样不仅浪费时间，还损耗精力。但要是合理分类，比如把课本放一起、漫画放一起，找起来就轻松多了。数据结构就像是你整理房间的方法。在编程里，数据结构决定了如何存储、组织和管理数据，让程序能高效地运行，就像有序的房间让生活更便捷。而这一章，便是我们启程的关键站点——数据结构的基础地带，重点聚焦链表、栈和队列。

本章将借助经典的竞赛真题，带领大家深入掌握链表、栈和队列这些数

据结构的基本用法。在解题过程中，你会真切感受到链表如何像灵活的书架布局，动态调整图书存放位置；栈怎样类似整理书籍时，处理复杂逻辑；队列又如何像图书馆借阅排队，有序调度资源。通过对这些真题抽丝剥茧般的分析，详细的代码实现演示，大家不仅能透彻理解各类数据结构在实际问题中的运用，更能举一反三，面对不同竞赛题目时，精准判断并选择最合适的数据结构来存储和处理数据，为在 CSP 竞赛中取得优异成绩筑牢根基。



1.1 链表——更好地插入和删除

本节学习目标

- ◆ 了解链表与数组的区别，能够对比链表与数组在插入、删除操作上的性能差异。
- ◆ 掌握链表的插入、删除和遍历操作，理解链表的基本操作和通用方法。
- ◆ 理解链表操作的时间复杂度，并能够解释其优势。
- ◆ 识别适合使用链表解决的问题场景，独立设计链表解决方案实际编程竞赛中的问题。
- ◆ 能够通过链表的应用，提升逻辑思维和问题解决能力。

1.1.1 问题引入：约瑟夫环

问题描述

n 个人围成一圈，从第一个人开始报数,数到 m 的人出列，再由下一个人重新从1开始报数，数到 m 的人再出圈，依次类推，直到所有的人都出圈，请输出依次出圈人的编号。

具体实例

假设有5个人（ $n=5$ ），每次报到3的人出列（ $m=3$ ），过程如下：

在第一轮中，从1开始报数，数到3时3出列，剩下[1, 2, 4, 5]；第二轮从4开始报数，数到3时1出列，剩下[2, 4, 5]；第三轮从2开始报数，数到3时5出列，剩下[2, 4]；第四轮从2开始报数，数到3时2出列，剩下[4]；最后一轮从4开始报数，数到3时4出列。最终出列顺序为[3, 1, 5, 2, 4]。

1.1.2 知识核心

我们来思考一下用数组解决这个问题的做法。

创建一个长度为 n 的数组，用数组元素来表示每个人，通过标记的方式来表示某人是否已经出圈。在报数过程中，不断遍历数组，当数到 m 时，将对应的数组元素标记为已出圈，然后继续从下一个未出圈的元素开始报数。

虽然数组可以解决这个问题，但它存在一些明显缺陷。在每次有人出圈后，后续的报数需要跳过已经出圈的人，这就需要不断地遍历数组，判断元素是否已经出圈，导致时间复杂度较高。而且，数组的插入和删除操作相对复杂，对于这种动态变化的问题，使用数组会显得不够灵活。这时候我们可以考虑用链表来解决这个问题。

在此，先回顾一下链表的重要知识。

链表的链式存储结构使得它在处理这种循环报数和删除节点的问题上具有天然的优势。在链表中，我们可以很方便地通过修改指针来实现节点的删除操作，而不需要像数组那样移动大量元素。

链表的特性在这里得到了充分的体现。

链式存储

顺序存储是指将数据元素依次存放在一块连续的存储空间中，就像在一排连续的房间里依次存放物品一样。顺序存储的优点是可以通过下标直接访问元素，访问速度快，时间复杂度为 $O(1)$ 。但缺点也很明显，插入和删除元素时可能需要移动大量元素。比如在数组中间插入一个元素，需要将插入位置之后的所有元素都向后移动一位，时间复杂度为 $O(n)$ 。

而**链式存储**是通过节点之间的指针连接来组织数据，每个节点可以存放在内存的任意位置，就像用绳子将分散在各处的物品串起来一样。链表中的节点通过指针相互关联，形成一个链式结构。链式存储的优点是插入和删除元素比较方便，只需要修改指针的指向即可，时间复杂度为 $O(1)$ （前提是已经知道要操作的节点位置）。缺点是不能随机访问元素，要访问链表中的某个节点，必须从链表的头节点开始，依次遍历链表，时间复杂度为 $O(n)$ 。

节点结构

链表每个节点包含**数据域**和**指针域**，数据域用于存储实际的数据，可以是任意类型的数据，比如整数、字符、结构体等，如题可以存储人的编号。指针域用于存储指向下一个节点的指针，通过这个指针可以将各个节点连接起来，形成链表。

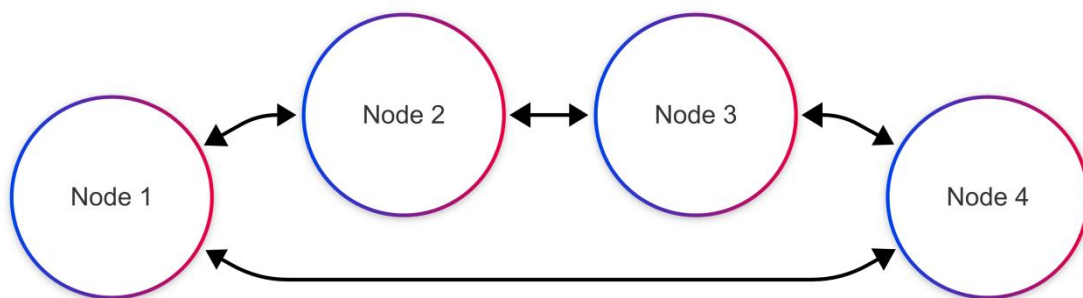


图1.1.1 指针域示意图

节点结构体是实现循环链表的要点，如示意图，循环由节点和指针构成的。我们正在实现的是双向链表，因此两个节点之间相互持有指向对方的指针。循环链表之所以叫循环，是因为最后一个节点与第一个节点连在了一起。通过使用循环双向链表，我们可以方便的模拟出题目要求的环形的报数。

下面是循环链表节点结构体定义的部分代码。

```
1 struct node
2 {
3     int val, nxt, pre;
4     //数据域-（双向）指针域
5 };
6
7 const int N = 120;
8 node t[N];
9 //使用数组写法便于调试，也有更好的常数优势（连续内存对cache友好）
```

图1.1.2 循环链表节点结构体定义的部分代码

首先，`struct node` 定义了一个节点结构体。这个结构体包含三个成员变量：数据域`val`、指针域`nxt`和`pre`，其中`nxt` 指针指向链表中的下一个节点，`pre` 指针指向链表中的前一个节点。这种双向指针的设计使得在链表中进行遍历和操作更加灵活，例如可以方便地向前或向后遍历链表，在删除节点时也更容易处理指针的更新。

其次，`const int N = 120`定义了一个常量 `N`，这个常量用于指定数组 `t` 的大小。
`node t[N];` 声明了一个类型为 `node` 的数组 `t`。

时间复杂度对比

在链表中删除一个节点的时间复杂度为 $O(1)$ ，而在数组中删除一个元素可能需要 $O(n)$ 的时间复杂度，因此链表在处理约瑟夫环问题时效率更高。由此我们可以对比出顺序存储和链式存储的时间复杂度差别。

操作	顺序存储（数组）	链式存储（链表）
插入（已知位置）	$O(n)$	$O(1)$
删除（已知位置）	$O(n)$	$O(1)$

图1.1.3 两种存储方式的时间复杂度对比

1.1.3 算法实现分步解析

步骤1：理解问题

已知约瑟夫环的问题是： n 个人围成一圈，从第1个人开始报数，每报到 m 的人出列，直到所有人出列。这个过程要求按出列顺序输出每个人的编号。刚刚我们假设 $n=5$ ， $m=3$ 时，出列顺序应为 $3 \rightarrow 1 \rightarrow 5 \rightarrow 2 \rightarrow 4$ 。

通过上述思考问题动手操作，我们会发现利用已有的数组知识来解决这个问题时，需要删除元素，这个过程需要移动大量数据，大大增加了时间复杂度（ $O(n)$ ），这个时候需要一种支持高效删除的数据结构。

步骤2：数据结构选择

为了高效处理环形结构，避免时间复杂度以及空间复杂度过高，我们采用一种新的数据结构——链表（如下定义）。

```

4 struct node
5 {
6     int val, nxt, pre;
7     // 数据域-（双向）指针域
8 };
9
10 const int N = 120;
11 node t[N]; // 使用数组写法便于调试，也有更好的常数优势（连续内存对cache友好）
12 int cnt = 1;

```

图1.1.4 链表定义的代码

以下是每个节点的定义以及作用：

val: 存储当前人的编号

nxt: 指向下一个人的数组下标

pre: 指向上一个人的数组下标

在这个过程中我们用数组来模拟链表，这样做的好处是内存连续，访问速度快，且无需动态分配内存。

步骤3：初始化链表

代码如图1.1.5所示：

```

19 int n, m;
20 cin >> n >> m;
21 for (int i = 1; i ≤ n; i++)
22 {
23     t[i] = {i, i + 1, i - 1};
24 }
25 t[1].pre = n;
26 t[n].nxt = 1; // 形成循环链表

```

图1.1.5 初始化链表的代码

假设n=5，初始化过程如下：

此时链表结构如下（箭头表示nxt方向）：

1 ↔ 2 ↔ 3 ↔ 4 ↔ 5 → 1（循环）

步骤4：模拟报数过程

代码如1.1.6图所示：

```

28     while (n-->0)
29     {
30         for (int i = 1; i <= m; i++)
31         {
32             cnt = t[cnt].next;
33         }
34         // 此时cnt指向的是第m + 1名同学
35         cnt = t[cnt].pre;
36         cout << t[cnt].val << " ";
37         int front = t[cnt].pre, rear = t[cnt].next;
38         t[front].next = rear;
39         t[rear].pre = front;
40
41         // 下一个报数的同学是第m + 1个
42         cnt = t[cnt].next;
43     }

```

图1.1.6 模拟报数的代码

其中，变量cnt表示当前报数的起点。每次循环将执行以下操作：

·子步骤4.1：找到第m个人

通过移动m次cnt，此时cnt指向第m+1个人。例如：

初始cnt=1，m=3时：

第1次移动：cnt=2（报数1）

第2次移动：cnt=3（报数2）

第3次移动：cnt=4（报数3，此时cnt指向第4人）

此时，实际要出列的是第3人（即cnt的前驱）。

·子步骤4.2：删除当前节点

以n=5，m=3为例：

第一次循环：删除3，链表变为1 ↔ 2 ↔ 4 ↔ 5 → 1，下一轮从4开始。

第二次循环：移动3次（4→5→1→2），删除1，链表变为2 ↔ 4 ↔ 5 → 2，下一轮从2开始。

依此类推，直到所有人出列。

步骤5：完整代码展示

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 struct node
5 {
6     int val, nxt, pre;
7     // 数据域- (双向) 指针域
8 };
9
10 const int N = 120;
11 node t[N]; // 使用数组写法便于调试, 也有更好的常数优势 (连续内存对cache友好)
12 int cnt = 1;
13
14 int main()
15 {
16     ios::sync_with_stdio(false);
17     cin.tie(nullptr);
18
19     int n, m;
20     cin >> n >> m;
21     for (int i = 1; i ≤ n; i++)
22     {
23         t[i] = {i, i + 1, i - 1};
24     }
25     t[1].pre = n;
26     t[n].nxt = 1; // 形成循环链表
27
28     while (n--)
29     {
30         for (int i = 1; i ≤ m; i++)
31         {
32             cnt = t[cnt].nxt;
33         }
34         // 此时cnt指向的是第m + 1名同学
35         cnt = t[cnt].pre;
36         cout << t[cnt].val << ' ';
37         int front = t[cnt].pre, rear = t[cnt].nxt;
38         t[front].nxt = rear;
39         t[rear].pre = front;
40
41         // 下一个报数的同学是第m + 1个
42         cnt = t[cnt].nxt;
43     }
44
45     return 0;
46 }

```

图1.1.7 完整代码



链表基础探索：单向链表（B3631）

实现一个数据结构，维护一张表（最初只有一个元素1）。需要支持下面的操作，其中 x 和 y 都是1到106范围内的正整数，且保证任何时间表中所有数字均不相同，操作数量不多于105：

- 1.将元素 y 插入到 x 后面；
- 2.询问 x 后面的元素是什么。如果 x 是最后一个元素，则输出0；
- 3.从表中删除元素 x 后面的那个元素，不改变其他元素的先后顺序。

输入格式：第一行一个整数 q 表示操作次数。接下来 q 行，每行表示一次操作，操作具体见题目描述。

输出格式：对于每个操作2，输出一个数字，用换行隔开。

1.1.4 深入讨论

时间复杂度

（1）访问元素

·链表中的元素是通过指针依次访问的，因此访问特定元素需要从头节点开始遍历，时间复杂度是 $O(n)$ 。

（2）插入元素

·如果已知要插入位置的前一个节点，则插入操作只需要调整几个指针，时间复杂度是 $O(1)$ 。

·如果不知道具体位置，则需要先遍历找到插入点，时间复杂度为 $O(n)$ 。

（3）删除元素

·如果已知要删除的节点，则删除操作只需要调整几个指针，时间复杂度是 $O(1)$ 。

·如果不知道具体位置，则需要先遍历找到删除点，时间复杂度为 $O(n)$ 。

空间复杂度

链表的空间复杂度是 $O(n)$ ，其中 n 是链表中节点的数量。

每个节点除了存储数据外，还需要存储指向下一个节点的指针（以及可能的前一个节点的指针，如果是双向链表）。

因此，链表相对于数组会有额外的指针存储空间开销。

易错点提示

下表中展示了我们在处理链表问题时容易出现的一些错误及相应的解决方法：

错误类型	后果	解决方法
空链表直接操作	程序崩溃	操作前检查 <code>if (head == nullptr)</code>
头尾节点更新遗漏	链表断裂或内存泄漏	传递引用或二级指针，记录前驱节点
单节点处理不当	野指针或逻辑错误	删除后显式置空头指针
双向链表指针不一致	链表结构破坏	插入/删除时同步更新 <code>prev</code> 和 <code>next</code>

图1.1.8 常见错误分析表

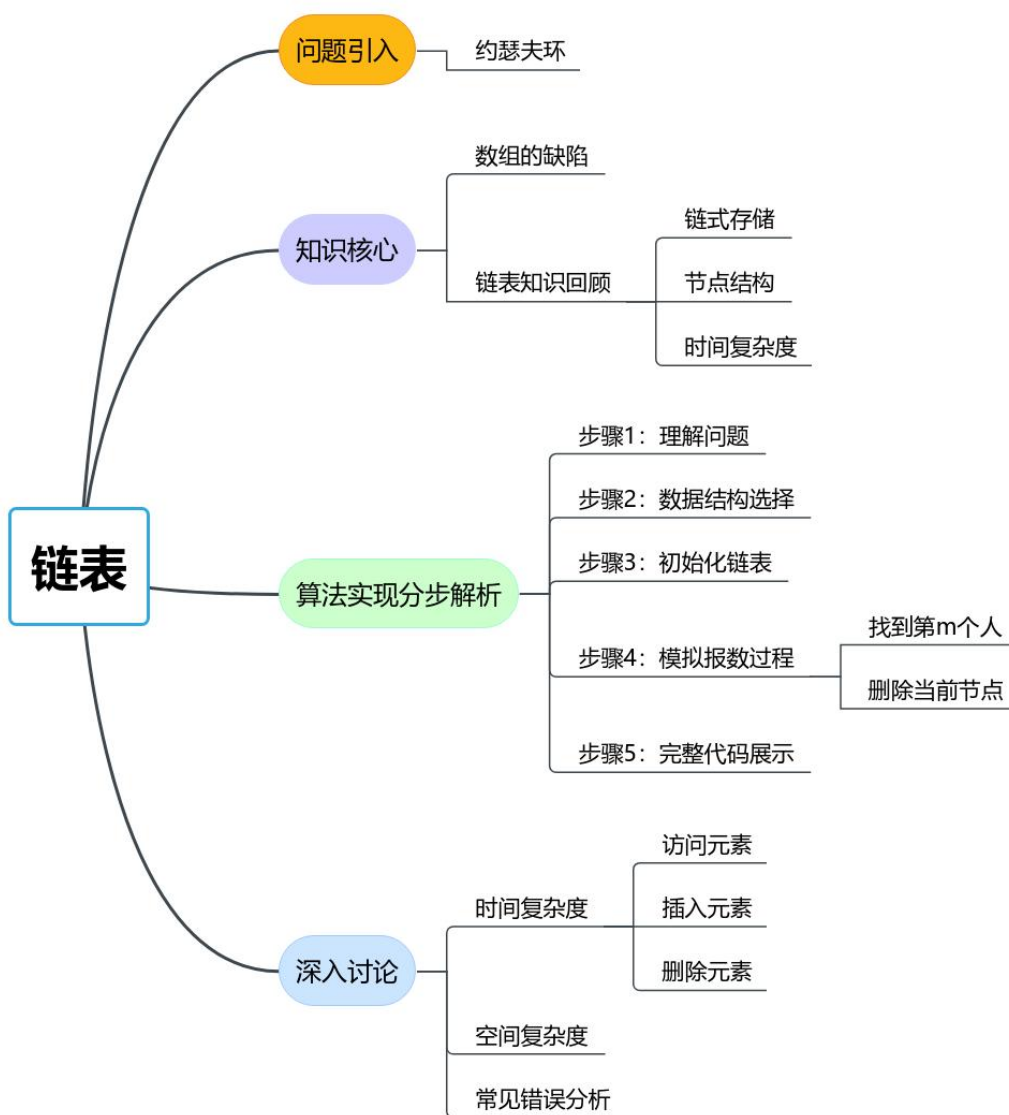


链表高级应用：邻值查找（P1046）

给定一个长度为 n 的序列 A ， A 中的数各不相同。对于 A 中的每一个数 A_i ，求： $\min_{1 \leq j < i} |A_i - A_j|$ 以及令上式取到最小值的 j （记为 P_i ）。若最小值点不唯一，则选择使 A_j 较小的那个。

输入格式：第一行输入整数 n ，代表序列长度。第二行输入 n 个整数 $A_1 \sim A_n$ ，代表序列的具体数值，数值之间用空格隔开。

输出格式：输出共 $n-1$ 行，每行输出两个整数，数值之间用空格隔开。分别表示当 i 取 $2 \sim n$ 时，对应的 $\min_{1 \leq j < i} |A_i - A_j|$ 和 P_i 的值。



练习提升

1. 编写程序，利用随机函数，生成 10 个 $0 \sim 10$ 的随机整数存储在数组中，并判断其中是否有数字“5”，若有，则输出它在数组中的下标（如有多个，也一并输出）；否则，输出“NO DATA”。
2. 编写程序，使用二维数组构造出以下的杨辉三角形（要求输出 n 行， $n > 9$ ）。
杨辉三角形是南宋数学家杨辉所著的《详解九章算术》一书中用三角形解释二项式系数的乘方规律，是二项式系数在三角形中的一种几何排列。

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮
```

1.2 栈——更好的对称匹配

本节学习目标

- ◆ 理解栈的基本概念与特性，能够对比栈与数组、链表在插入、删除操作上的性能差异。
- ◆ 掌握栈的基本操作，并能够用数组或链表模拟实现栈结构。
- ◆ 理解栈操作的时间复杂度，并能够解释其优势。
- ◆ 识别适合栈解决的问题场景，独立设计基于栈的解决方案。

1.2.1 问题引入：闭合判别

问题描述

给定一个仅包含 `()` 和 `[]` 的字符串，判断括号是否合法闭合。

具体实例

若输入：`()[] ([])`，则输出：YES；若输入：`([] ([])`，则输出：NO。

1.2.2 问题拆解

合法闭合

在解决这个问题之前，我们先来回顾一下**合法闭合**这个关键概念。

在编程中，特别是在处理表达式求值和括号匹配问题时，合法闭合是至关重要的。**合法闭合**指的是按照正确的顺序关闭或配对括号的过程。例如：在算术表达式中，每个左括号“`(`”都需要有一个对应的右括号“`)`”来闭合它，反之亦然。合法闭合确保了表达式的结构正确性和逻辑一致性。

那开动咱们聪明的小脑筋，如果有多个括号，我们该如何快速判断括号的嵌套关系呢？这里提示一个关键点：在括号匹配过程中，最后出现的左括号需要最优先匹配。很矛盾是不是？那根据这个关键点是否能想出相关的数据结构呢？

栈

不错，想必大家已经想到了，这可以通过使用“栈”这种数据结构来实现。栈是一种“后进先出”（LIFO）的线性数据结构。

每当遇到一个左括号时，就将其推入栈中；每当遇到一个右括号时，就检查栈顶是否有对应的左括号。如果有，就将左括号从栈中弹出，表示一对括号成功匹配。这也代表着最后出现的左括号就是最后一个被推入栈中的（此时是栈顶，好比图1.2.1盘子堆的最顶上的盘子，在堆放盘子时它是最后一个放上去的），同时也是第一个从栈中弹出的（从一摞盘子中拿出一个盘子，都是从最顶上第一个开始拿）。



图1.2.1 一摞盘子

但“后进先出”（LIFO）也只是栈的一个主要特征，我们一起来看看栈的其他基本操作吧。

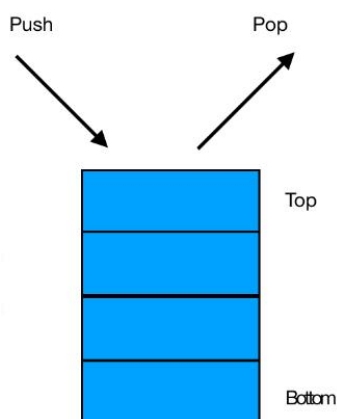


图1.2.2 后进先出操作示意图

栈的基本操作

栈的基本操作主要包括四点，如下表：

操作	作用
push ()	将元素压入栈
pop ()	弹出栈顶元素
top ()	获取栈顶元素
bottom ()	获取栈底元素
size ()	统计栈中元素数量

1.2.3 用数组模拟栈

在之前我们回顾了栈的基本用法，知道栈是一种后进先出（Last In First Out, LIFO）的数据结构。接下来，我们来看看如何用数组来模拟栈的操作。

首先，我们需要定义一个数组和一个指针来模拟栈。数组用来存储栈中的元素，指针用来记录栈顶的位置。在这里，N 定义了我们模拟栈的最大容量为 1000，st 数组就是我们的栈空间，而 ptr 是栈顶指针，初始值为 0，表示栈为空，它指向的是下一个可以插入元素的位置。

```
const int N = 1000;  
int st[N]; // 栈空间  
int ptr = 0; // 栈顶指针（指向下一个空位）
```

图1.2.3 定义数组和指针

接下来，我们实现栈的几个基本操作：

- **压栈操作（push）**：将元素放入栈中。

在 push 函数中，我们先将 ptr 指针自增 1，让它指向下一个空位，然后把要压入栈的元素 x 存储在 st[ptr] 的位置上。这样就完成了一次压栈操作。

```
void push(int x) { st[++ptr] = x; }
```

图1.2.4 压栈操作

- **弹栈操作 (pop)**：从栈中取出元素。

pop 函数用于弹出栈顶元素。我们先检查 ptr 是否大于 0，如果大于 0，说明栈中至少有一个元素，此时将 ptr 指针减 1，相当于把栈顶元素移除了。如果 ptr 等于 0，说明栈为空，不进行任何操作。

```
void pop() { if (ptr > 0) ptr--; }
```

图1.2.5 弹栈操作

- **获取栈顶元素 (top)**：查看栈顶的元素但不弹出它。

top 函数直接返回 st[ptr] 的值，也就是栈顶元素的值。不过要注意，在调用这个函数之前最好先确保栈不为空，否则返回的结果是没有意义的。

```
int top() { return st[ptr]; }
```

图1.2.6 获取栈顶元素

- **获取栈的大小 (size)**：了解栈中当前元素的数量。

size 函数很简单，直接返回 ptr 的值，因为 ptr 记录的是栈中元素的个数（同时也是下一个空位的位置）。

```
int size() { return ptr; }
```

图1.2.7 获取栈的大小

1.2.4 算法实现分步解析

步骤1：理解问题

已知问题要求是给定一个仅由 ()、[] 组成的字符串，判断其是否合法。这个时候我们思考什么情况下这个字符串是合法，即合法的条件。首先字符串要满足

闭合性——每个右括号必须有对应的左括号。其次要满足顺序性——括号必须按正确顺序闭合（例如 "[D]" 不合法，因为 D），闭合时最近的左括号是 []）。

■ 示例：

合法："[()]"、"([)]"。

非法："[D]"（顺序错）、"([)"（未闭合）、"]()"（右括号未匹配左括号）。

步骤2：数据结构选择

通过问题分析，我们发现这个问题的核心需求是需要快速找到当前右括号对应的左括号，且必须是最新未闭合的左括号。这个问题解决过程符合栈的特性，即**后进先出**（Last In First Out）。后进入栈的左括号会先被匹配，完美契合括号的闭合顺序。

接下来我们来对比其他结构：

数组/链表：需要额外指针记录最近左括号位置，不如栈直接。

哈希表：无法直接处理顺序问题。

综上所述，栈是解决此类“对称匹配问题”的最优数据结构。

步骤3：栈的初始化

代码如图1.2.8所示：

```

4 const int N = 1000;
5 char s[N];
6 int ptr = 0;
7
8 void push(char x)
9 {
10     s[++ptr] = x;
11 }
12
13 void pop()
14 {
15     ptr--;
16 }
17
18 char top()
19 {
20     return s[ptr];
21 }
22
23 int size()
24 {
25     return ptr;
26 }

```

图1.2.8 初始化栈的代码

用字符数组 `s[N]` 模拟栈，`ptr` 是栈顶指针（初始为0，表示空栈）。

■ 关键操作：

`push(x)`：将 `x` 压入栈顶（先让 `ptr` 加1，再存入 `x`）。

`pop()`：栈顶指针减1（相当于删除栈顶元素）。

`top()`：返回当前栈顶元素（即 `s[ptr]`）。

`size()`：返回栈中元素个数（即 `ptr` 的值）。

步骤4：遍历字符串处理括号

代码如图所示：


```

28 int main()
29 {
30     ios::sync_with_stdio(false);
31     cin.tie(nullptr);
32     string str;
33     cin >> str;
34     bool flag = true;
35     for (auto c : str)
36     {
37         if (c == '(' || c == '[')
38         {
39             push(c);
40         }
41         else if (c == ')' && top() == '(')
42         {
43             pop();
44         }
45         else if (c == ']' && top() == '[')
46         {
47             pop();
48         }
49         else
50         {
51             flag = false;
52             break;
53         }
54     }
55     if (size())
56         flag = false;
57     cout << flag;
58     return 0;
59 }

```

图1.2.9 遍历字符处理括号代码

遍历字符串的每个字符 c:

- **子步骤4.1: 处理左括号 ((或 [)**

直接压入栈中，等待后续匹配。

例子：输入 "[("，栈会依次存入 "(" 和 "["。

- **子步骤4.2: 处理右括号) 或])**

检查栈是否为空：如果栈为空（比如输入 "]"），说明没有对应的左括号，标记非法（flag = false）。

检查栈顶是否匹配：

如果 c 是) 且栈顶是 (，弹出栈顶。

如果 c 是] 且栈顶是 [，弹出栈顶。

例子：输入 "()", 遇到) 时栈顶是 (, 匹配后栈变空。

不匹配的情况：如果栈顶不匹配（比如 c 是] 但栈顶是 (），标记非法。

步骤5：完整代码展示

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 const int N = 1000;
5 char s[N];
6 int ptr = 0;
7
8 void push(char x)
9 {
10     s[++ptr] = x;
11 }
12
13 void pop()
14 {
15     ptr--;
16 }
17
18 char top()
19 {
20     return s[ptr];
21 }
22
23 int size()
24 {
25     return ptr;
26 }
27
28 int main()
29 {
30     ios::sync_with_stdio(false);
31     cin.tie(nullptr);
32     string str;
33     cin >> str;
34     bool flag = true;
35     for (auto c : str)
36     {
37         if (c == '(' || c == '[')
38         {
39             push(c);
40         }
41         else if (c == ')' && top() == '(')
42         {
43             pop();
44         }
45         else if (c == ']' && top() == '[')
46         {
47             pop();
48         }
49         else
50         {
51             flag = false;
52             break;
53         }
54     }
55     if (size())
56         flag = false;
57     cout << flag;
58     return 0;
59 }
```

图1.2.10 完整代码

■ 注意：

遍历后栈必须为空！（if (size() != 0) flag = false;）如果栈不为空，强制标记为非法。

1.2.5 扩展应用：栈的典型场景

我们成功运用数组模拟栈的方法，巧妙解决了括号匹配问题，对栈的基本操作与应用有了扎实掌握。接下来我们将目光投向一类非常经典且更具挑战性的问题——利用栈求表达式的值。

请思考一下如何求出表达式 $3 * (2 + 5)$ 的值。

表达式求值的关键在于**处理运算符的优先级**。对于加、减、乘、除以及括号，乘除运算优先级高于加减，而括号内的运算又优先于括号外。为了准确计算，我们需要按正确顺序处理运算符和操作数，因此我们会用到两个栈，一个用于存储操作数，另一个用于存储运算符。

（1）扫描表达式

先从左到右逐个扫描表达式中的字符。

（2）处理操作数

当遇到数字时，将其解析为完整的数值，然后压入操作数栈。例如，遇到3，就把3压入操作数栈；遇到2和5时，也依次压入。

（3）处理运算符

当遇到左括号“（”时，直接将其压入运算符栈。这是因为左括号标志着一个子表达式的开始，后续遇到的运算符和操作数会在这个子表达式内处理。

当遇到运算符（如+、-、*、/）时，需要与运算符栈顶的运算符比较优先级。若当前运算符优先级高于栈顶运算符，就将当前运算符压入运算符栈；若当前运算符优先级低于或等于栈顶运算符，就从操作数栈中弹出两个操作数，从运算符栈中弹出一个运算符进行计算，然后把计算结果压回操作数栈，接着继续比较当前运算符与新的栈顶运算符的优先级，重复上述过程，直到当前运算符可以压入运算符栈。

(4) 最终计算

扫描完整个表达式后，运算符栈中可能还残留运算符。此时，依次从操作数栈弹出两个操作数，从运算符栈弹出一个运算符进行计算，将结果压回操作数栈，直到运算符栈为空。此时，操作数栈中剩下的唯一元素就是整个表达式的计算结果。

以下是完整代码：

```
1 #include <iostream>
2 #include <stack>
3 #include <string>
4 #include <cctype>
5
6 // 获取运算符优先级
7 int precedence(char op) {
8     if (op == '+' || op == '-') return 1;
9     if (op == '*' || op == '/') return 2;
10    return 0;}
11
12 // 执行运算
13 int applyOp(int a, int b, char op) {
14     switch (op) {
15         case '+': return a + b;
16         case '-': return a - b;
17         case '*': return a * b;
18         case '/':
19             if (b == 0) throw std::runtime_error("Division by zero");
20             return a / b;
21     }
22     return 0;}
23
24 // 表达式求值
25 int evaluate(const std::string& tokens) {
26     std::stack<int> values;
27     std::stack<char> ops;
28
29     for (size_t i = 0; i < tokens.length(); i++) {
30         if (tokens[i] == ' ') continue;
31
32         if (isdigit(tokens[i])) {
33             int val = 0;
34             while (i < tokens.length() && isdigit(tokens[i])) {
35                 val = (val * 10) + (tokens[i] - '0');
36                 i++;
37             }
38             i--;
39             values.push(val);
40         } else if (tokens[i] == '(') {
41             ops.push(tokens[i]);
```

图1.2.11 求表达式的值代码（上）

```

42     } else if (tokens[i] == ')') {
43         while (!ops.empty() && ops.top() != '(') {
44             int val2 = values.top();
45             values.pop();
46             int val1 = values.top();
47             values.pop();
48             char op = ops.top();
49             ops.pop();
50             values.push(applyOp(val1, val2, op));
51         }
52         if (!ops.empty()) ops.pop();
53     } else {
54         while (!ops.empty() && precedence(ops.top()) ≥ precedence(tokens[i])) {
55             int val2 = values.top();
56             values.pop();
57             int val1 = values.top();
58             values.pop();
59             char op = ops.top();
60             ops.pop();
61             values.push(applyOp(val1, val2, op));
62         }
63         ops.push(tokens[i]);
64     }
65 }
66
67 while (!ops.empty()) {
68     int val2 = values.top();
69     values.pop();
70
71     return 0;
}

```

图1.2.12 求表达式的值代码（下）

1.2.6 单调栈

问题描述

n 个人正在排队进入一个音乐会。人们等得很无聊，于是他们开始转来转去，想在队伍里寻找自己的熟人。

队列中任意两个人 a 和 b ，如果他们是相邻或他们之间没有人比 a 或 b 高，那么他们是可以互相看得见的。

写一个程序计算出有多少对人可以互相看见。

对于全部的测试点，保证 $1 \leq \text{每个人的高度} < 2^{31}$ ， $1 \leq n \leq 5 \times 10^5$ 。

问题分析

a.暴力做法：对于每对人检查中间是否有比他们高的，时间复杂度为 $O(n^2)$ ，显然无法通过本题。

b.优化思路：想象所有人在你面前站成一排，观察：对于队伍中的一个人 i ，他能看到的人有什么规律？



图1.2.13 排队示意图

他们的身高具有**单调性**。并且，这种单调性可以在枚举 i 的过程中维护。

比方说，你在第 i 个人处已经得到了前 $i-1$ 个人构成的“能看到的人序列”，这个序列的长度就是第 i 个人向左看能看到的人数。

现在你怎样计算第 $i+1$ 个人向左看能看到的人数呢？

这不难，只要把第 i 个人加入序列维护出答案就好。如果第 i 个人比序列最右边的人高，就把序列最右的人排除，因为他之后会被第 i 人挡住。请注意被排除的人很可能不止一个人。

每个人都只会被枚举一次，最多被加入序列一次，最多被排除一次，因此这种做法的时间复杂度是 $O(n)$ 。

怎么实现呢？你又注意到我们需要这个序列的大小，还需要对这个序列的最右端做操作。现在你联想到了什么数据结构？栈。

代码实现

■ 重点：

- (1) 使用栈未判空导致的越界
- (2) 边界条件：多个身高相同的人

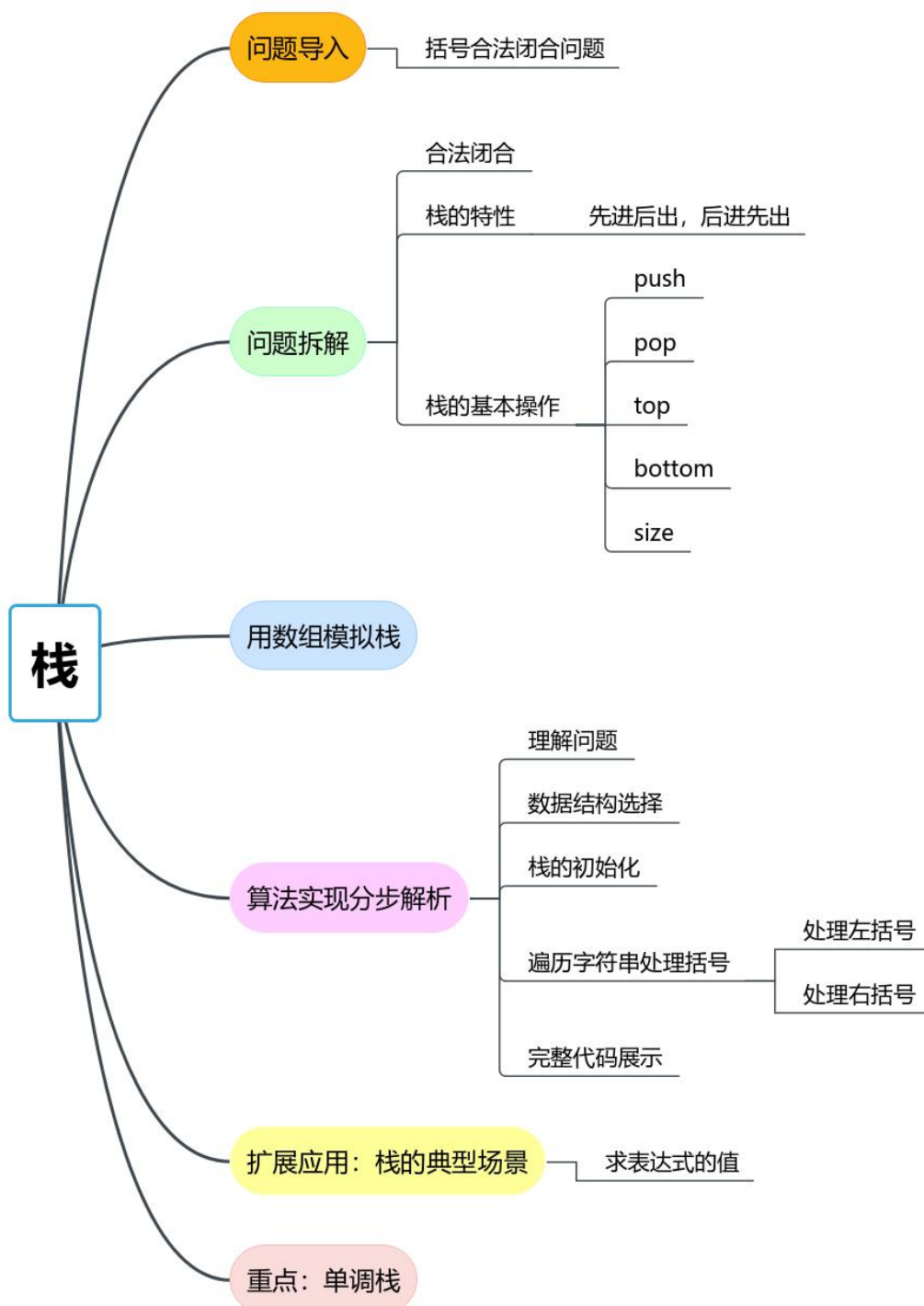
```

1 #include <bits/stdc++.h>
2 using namespace std;
3 using ll = long long;
4 const int N = 5e5 + 10;
5 using person = pair<ll, int>;
6 person st[N];
7 int ptr = 0;
8
9 void push(person x)
10 {
11     st[++ptr] = x;
12 }
13 void pop()
14 {
15     if (ptr > 0)
16         ptr--;
17 }
18 auto top()
19 {
20     return st[ptr];
21 }
22 int size()
23 {
24     return ptr;
25 }
26
27 int main()
28 {
29     ios::sync_with_stdio(false);
30     cin.tie(nullptr);
31
32     ll n, ans = 0;
33     cin >> n;
34     for (int i = 1; i ≤ n; i++)
35     {
36         ll height;
37         cin >> height;
38         person cnt = {height, 1};
39
40         while (size() && top().first ≤ height)
41         {
42             if (top().first == height)
43                 cnt.second += top().second;
44             ans += top().second;
45             pop();
46         }
47
48         if (size())
49             ans++;
50
51         push(cnt);
52     }
53     cout << ans;
54     return 0;
55 }
56

```

图1.2.14 单调栈完整代码

这样的技巧就被称为**单调栈**。





练习提升

1. 编写程序，利用栈将从键盘输入的十进制整数转换为十六进制数，并输出。
2. 一般情况下，求自然数 n 的阶乘 $n!$ ，需要知道 $(n-1)!$ ，依此类推，思考这一解题思路中是否用到栈的原理，小组讨论，说明理由。

1.3 队列——有序的先进先出

本节学习目标

- ◆ 理解队列的基本概念与特性。
- ◆ 掌握队列的基本操作，并能够解决队列相关问题。
- ◆ 理解队列操作的时间复杂度，并能够解释其优势。
- ◆ 识别适合队列解决的问题场景，独立设计基于队列的解决方案。
- ◆ 能够通过队列的应用，提升计算机思维和问题解决能力。

1.3.1 情境引入：滑动窗口

问题描述

有一个长为 n 的序列 a ，以及一个大小为 k 的窗口。现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

具体实例

对于序列 $[1,3,-1,-3,5,3,6,7]$ 以及 $k=3$ ，有如下过程：

窗口位置								最小值	最大值
[1	3	-1]	-3	5	3	6	7	-1	3
1	[3	-1	-3]	5	3	6	7	-3	3
1	3	[-1	-3	5]	3	6	7	-3	5
1	3	-1	[-3	5	3]	6	7	-3	5
1	3	-1	-3	[5	3	6]	7	3	6
1	3	-1	-3	5	[3	6	7]	3	7

图1.3.1 滑动窗口操作示意图

输入格式：

输入一共有两行，第一行有两个正整数 n, k 。 第二行 n 个整数，表示序列 a 。例如：

```
8 3
1 3 -1 -3 5 3 6 7
```

图1.3.2 滑动窗口输入示例

输出格式：

输出共两行，第一行为每次窗口滑动的最小值，第二行为每次窗口滑动的最大值。例如：

```
-1 -3 -3 -3 3 3
3 3 5 5 6 7
```

图1.3.3 滑动窗口输出示例

1.3.2 队列基础概念与实现

核心特性FIFO

队列（**queue**）是一种具有「先进入队列的元素一定先出队列」性质的表。由于该性质，队列通常也被称为先进先出（**first in first out**）表，简称 **FIFO**。

为了更好地理解队列的特性，我们可以把队列类比成地铁安检通道。在安检通道里，先进入通道的乘客会先完成安检，后进入的乘客则需要排在后面排队等待，依次接受安检，这和队列中的元素进出顺序是完全一致的。



图1.3.4 队列示意图

STL基本操作

在实际编写代码时，我们不需要自己去实现队列这种数据结构，C++ 标准模板库（STL）已经为我们提供了功能完善的 `std::queue`，我们只需要学会如何使用它就好。接下来，让我们看看 `std::queue` 常用的基本操作：

- **`queue<int> q;`**

此语句定义了一个名为 `q` 的队列对象，这个队列能够存储 `int` 类型的元素。这里的 `queue` 实际上是 `std::queue` 的简写，不过要保证在代码开头含 `<queue>` 头文件。

- **`q.push();`**

`push` 是 `std::queue` 的成员函数，其作用是把一个元素添加到队列的尾部。例如 `q.push(1)` 是将整数 1 加入到队列 `q` 里。

- **`q.pop();`**

`pop` 函数用于移除队列头部的元素。不过需要注意的是，在调用 `pop` 之前，要先检查队列是否为空，因为对空队列调用 `pop` 会引发未定义行为。

- **`q.front();`**

`front` 函数会返回队列头部元素的引用。同样，在调用 `front` 之前，也要确保队列不为空，不然会出现未定义行为。

- **`q.size();`**

`size` 函数返回队列中元素的数量。

- **`q.empty();`**

`empty` 函数用来判断队列是否为空。若队列为空，返回 `true`；反之，返回 `false`。

STL双端队列

除了 `std::queue`，STL中还有一种与之相关的数据结构 —— 双端队列（`deque`）。双端队列 `deque` 支持在 $O(1)$ 时间复杂度内进行首尾插入和删除操作。它和 `queue` 最大的区别在于，`deque` 既可以从队首进（出）队，又可以从队尾进（出）队，使用起来更加灵活。下面是`deque`的一些常见操作示例：

- **`deque<int> dq;`**

这行代码定义了一个名为 `dq` 的双端队列对象，该队列可以存储 `int` 类型的元素。这里的 `deque` 实际上是 `std::deque` 的简写，不过要确保在代码的开头包含 `<deque>` 头文件。

- `dq.push_back();`

`push_back` 是 `std::deque` 的成员函数，其作用是将一个元素添加到双端队列的尾部。例如 `dq.push_back(2)`，把整数 2 加入到双端队列 `dq` 的尾部。

- `dq.pop_front();`

`pop_front` 函数用于移除双端队列头部的元素。同样，在调用 `pop_front` 之前，需要先检查双端队列是否为空，因为对空的双端队列调用 `pop_front` 会导致未定义行为。

1.3.3 滑动窗口实现

步骤1：理解问题

已知问题要求：给定一个长度为 `n` 的序列 `a` 和一个大小为 `k` 的窗口，窗口从最左端开始向右滑动，每次滑动一个单位，输出每次窗口滑动后的最小值和最大值。

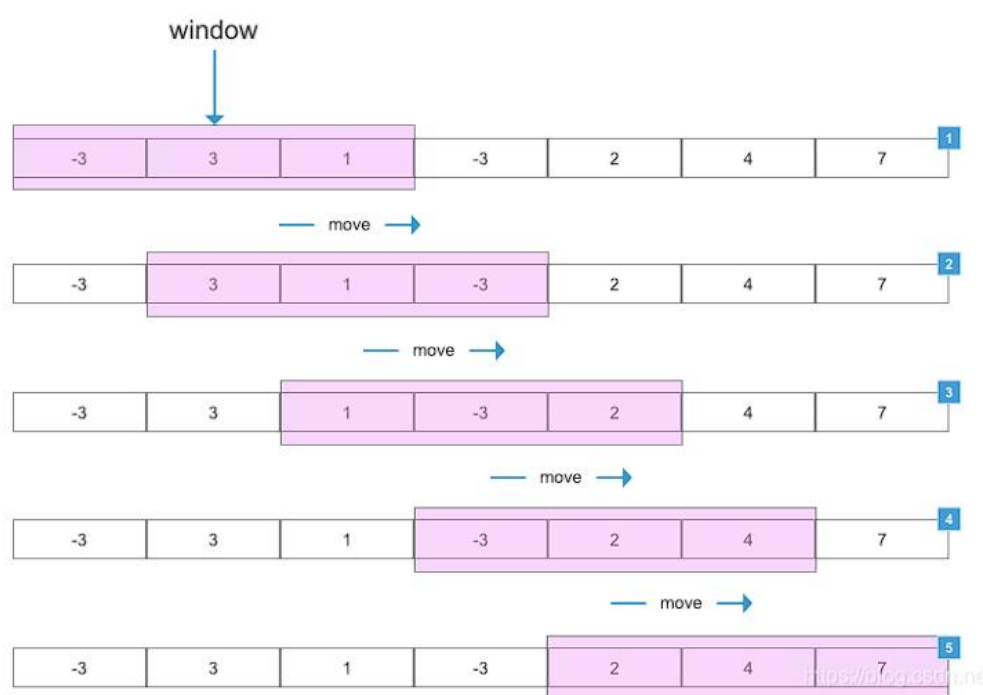


图1.3.5 滑动窗口

示例分析：

输入序列为 [1,3,-1,-3,5,3,6,7]，窗口大小 $k=3$ ，输出最小值序列为 -1,-3,-3,-3,3,3，最大值序列为 3,3,5,5,6,7。

关键约束：需在 $O(n)$ 时间复杂度内完成，避免暴力枚举的低效操作。

步骤2：数据结构选择

通过问题分析，发现核心需求是快速维护窗口范围，并高效获取极值。传统队列无法直接满足需求，因此引入**单调队列**：

- 单调递增队列：维护窗口最小值，队首始终为当前窗口最小值的索引。
- 单调递减队列：维护窗口最大值，队首始终为当前窗口最大值的索引。

对比其他结构：

- 数组/链表：遍历窗口内元素求极值的时间复杂度为 $O(k)$ ，无法满足高效性。
- 优先队列（堆）：虽然能快速获取极值，但无法高效删除滑动过程中移出窗口的元素。

同时，单调队列通过动态维护队列的单调性，可在 $O(1)$ 时间内获取极值，并保证操作整体时间复杂度为 $O(n)$ ，是最优选择。

步骤3：队列初始化与维护规则

关键操作逻辑（以最小值为例）：

- 初始化双端队列 `deque<int> dq`，存储元素索引。
- 遍历序列：对每个元素 `a[i]`，执行以下操作：
 - 移除过期元素：若队首索引 `dq.front() ≤ i - k`，说明该元素已不在窗口内，弹出队首。
 - 维护单调性：从队尾开始，若当前元素 `a[i] ≤ a[dq.back()]`，则弹出队尾元素，直到队列恢复单调递增。
 - 插入当前索引：将 `i` 加入队尾。

- 记录结果：当 $i \geq k-1$ 时，队首元素即为当前窗口的最小值。

最大值队列的逻辑与最小值类似，只需将比较条件改为 $a[i] \geq a[dq.back()]$ 。

代码如1.3.2图所示：

```
vector<int> min_ans, max_ans;
deque<int> dq;

// 处理最小值队列（递增）
for (int i = 0; i < n; ++i) {
    while (!dq.empty() && dq.front() ≤ i - k) {
        dq.pop_front();
    }
    while (!dq.empty() && a[i] ≤ a[dq.back()]) {
        dq.pop_back();
    }
    dq.push_back(i);
    if (i ≥ k - 1) {
        min_ans.push_back(a[dq.front()]);
    }
}
```

图1.3.5 处理最小值

步骤4：完整代码展示

```
1 #include <iostream>
2 #include <vector>
3 #include <deque>
4 using namespace std;
5
6 int main() {
7     ios::sync_with_stdio(false);
8     cin.tie(nullptr);
9
10    int n, k;
11    cin >> n >> k;
12    vector<int> a(n);
13    for (int i = 0; i < n; ++i) {
14        cin >> a[i];
15    }
16
17    vector<int> min_ans, max_ans;
18    deque<int> dq;
19
20    // 处理最小值队列（递增）
21    for (int i = 0; i < n; ++i) {
22        while (!dq.empty() && dq.front() ≤ i - k) {
23            dq.pop_front();
24        }
25        while (!dq.empty() && a[i] ≤ a[dq.back()]) {
26            dq.pop_back();
27        }
28        dq.push_back(i);
29        if (i ≥ k - 1) {
30            min_ans.push_back(a[dq.front()]);
31        }
32    }
```

图1.3.6 完整代码（上）

```

35
36 // 处理最大值队列 (递减)
37 for (int i = 0; i < n; ++i) {
38     while (!dq.empty() && dq.front() ≤ i - k) {
39         dq.pop_front();
40     }
41     while (!dq.empty() && a[i] ≥ a[dq.back()]) {
42         dq.pop_back();
43     }
44     dq.push_back(i);
45     if (i ≥ k - 1) {
46         max_ans.push_back(a[dq.front()]);
47     }
48 }
49
50 // 输出结果
51 for (int i = 0; i < min_ans.size(); ++i) {
52     if (i) cout << " ";
53     cout << min_ans[i];
54 }
55 cout << '\n';
56
57 for (int i = 0; i < max_ans.size(); ++i) {
58     if (i) cout << " ";
59     cout << max_ans[i];
60 }
61 cout << '\n';
62
63 return 0;
64 }

```

图1.3.6 完整代码（下）

易错点提示

- **错误原因：**直接调用`q.front()`，当队列为空时，这会引发问题。
- **解决方法：**在调用`front()`或`pop()`之前，先用`empty()`方法检查队列是否为空。

```

1 // 错误示例：未检查队列空直接访问
2 int val = q.front();
3 // q为空时导致运行时错误
4 // 正确写法
5 if(!q.empty())
6 {
7     val = q.front();
8     q.pop();
9 }

```

图1.3.7 队列越界问题

1.3.5 总结与扩展应用：BFS广度优先搜索

题目描述

马的遍历（P1443）：

有一个 $n \times m$ 的棋盘，在某个点 (x,y) 上有一个马，要求你计算出马到达棋盘上任意一个点最少要走几步。对于全部的测试点，保证 $1 \leq x \leq n \leq 400$ ， $1 \leq y \leq m \leq 400$ 。

输入输出

• 输入格式

输入只有一行四个整数，分别为 n,m,x,y 。




```
3 3 1 1
```

图1.3.8 输入范例

• 输出格式

一个 $n \times m$ 的矩阵，代表马到达某个点最少要走几步（不能到达则输出 -1）



0	3	2
3	-1	1
2	1	4

图1.3.9 输出范例

核心思路

（1）问题建模

将棋盘视为一个无权图，每个格子是图的节点。马从起点出发，每次按“日”字形移动（8种方向），求到达所有节点的最短路径。

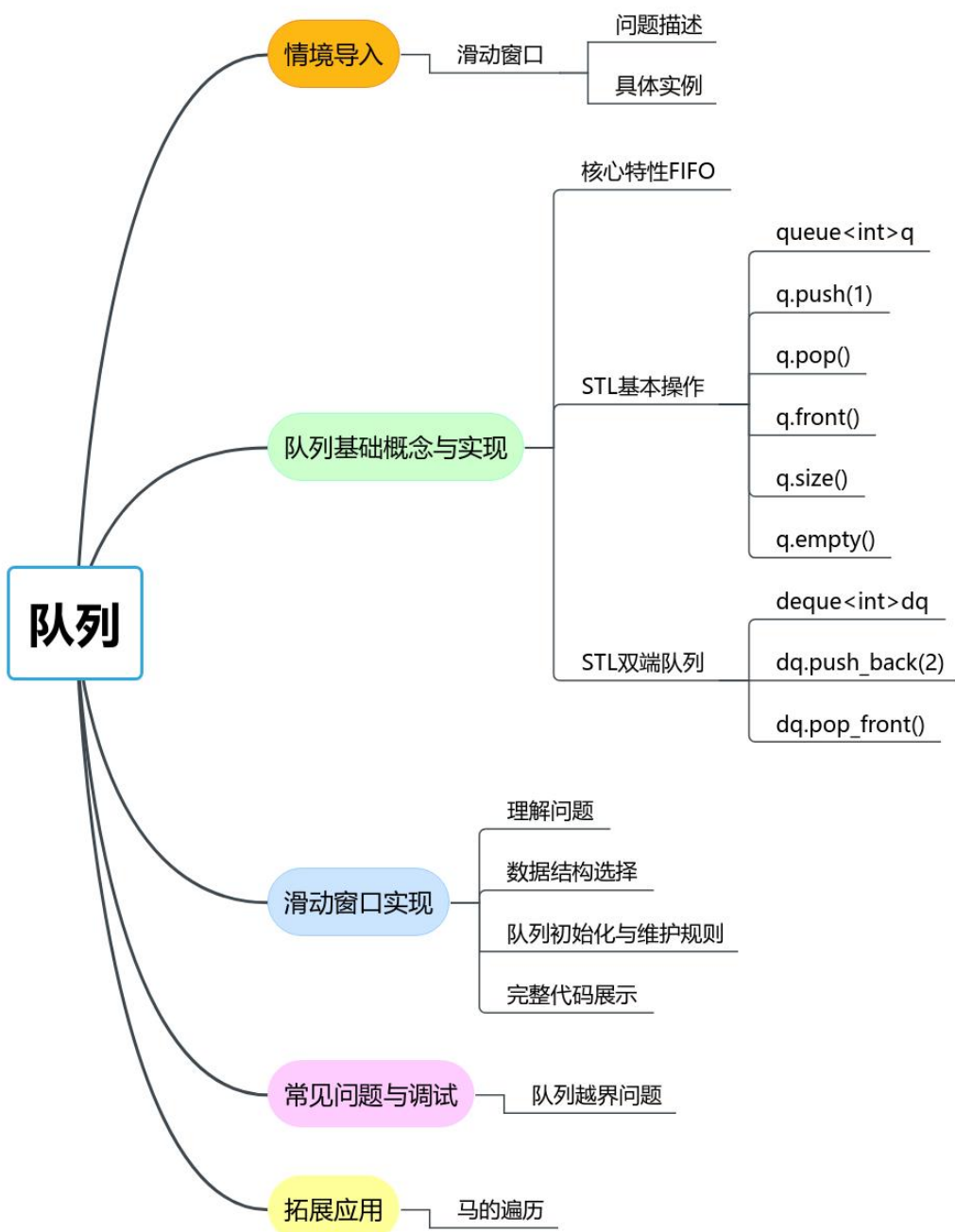
（2）BFS 的适用性

BFS 按层遍历，首次访问到某节点时所用的步数即为最短步数，天然适合解决单源最短路径问题。

核心代码展示

```
1  queue<pt> q;  
2  q.push({x, y});  
3  int cnt = 0;  
4  auto check = [n, m, &dis](pt a) {  
5      if (a.x > n || a.x < 1 || a.y > m || a.y < 1)  
6          return false;  
7      if (dis[a.x][a.y] == inf)  
8          return true;  
9      return false;  
10 };  
11 dis[x][y] = 0;  
12 while (!q.empty())  
13 {  
14     auto now = q.front();  
15     q.pop();  
16     auto [a, b] = now;  
17     cnt = dis[a][b];  
18     for (int j = 0; j < 8; j++)  
19         if (check({a + X[j], b + Y[j]}))  
20             q.push({a + X[j], b + Y[j]}), dis[a + X[j]][b + Y[j]] = cnt + 1;  
21 }
```

图1.3.10 马的遍历问题核心代码展示

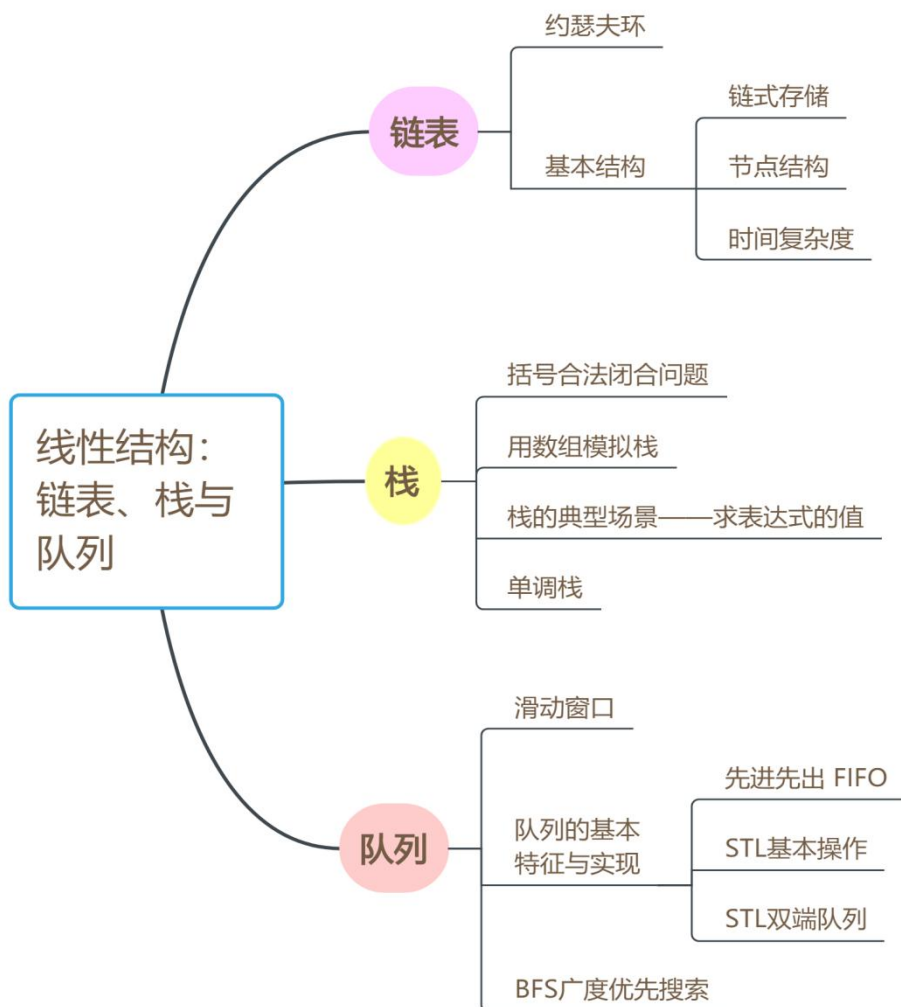




练习提升

1. 假设用长度为 11 的数组作为顺序队列，初始为空。分别绘出做完下列操作后的队列示意图。若有元素不能入队，试说明理由。操作序列如下：
(1) A, B, C, D, E 依次入队； (2) A, B 依次出队；
(3) F, G, H, I, J 依次入队； (4) O, P, Q, R 依次入队。
2. 利用两个顺序栈 S1 和 S2 模拟一个队列。利用栈的基本操作，实现队列的入队和出队操作，并说出基本思路。

1. 下图展示了本章的核心概念与关键能力，请同学们对照图中的内容进行总结。



2. 根据自己的掌握情况填写下表。

学习内容	掌握程度
链表的基本结构	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
链表的应用	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
用数组模拟栈	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
栈的典型场景应用	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
队列的基本特征与实现	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
队列的应用	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解

小结

在本章中，我们围绕链表、栈和队列这三种基础且核心的数据结构展开了深入学习。通过经典竞赛真题的实战演练，我们不仅掌握了这些数据结构的基本用法，更学会从实际问题出发，依据数据特点和操作需求，精准选择合适的数据结构。这不仅提升了我们解决问题的效率，更是理解高级数据结构与复杂算法的重要铺垫。希望大家能将本章所学融会贯通，为后续 CSP-J 竞赛征程夯实基础，在面对各类难题时，灵活运用这些数据结构，展现扎实的编程功底与卓越的算法思维。